System and Method for Reducing Memory Leaks in Virtual Machine Programs

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates in general to a system and method for reducing memory leaks in virtual machine programs. More specifically, the present invention relates to a system and method for using a compiler to add nullification statements in a program.

2. Description of the Related Art

Middleware environments, such as that provided by a Java Virtual Machine (JVM), often use a heap to store objects created in the environment. The heap stores objects created by an application running in the middleware environment. Garbage collection is the process of automatically freeing objects from the heap that are no longer needed by the application. This reduces the burden of memory management placed on the programmer.

However, garbage collection is not a panacea. Long running middleware applications remain susceptible to failures induced by "memory leaks." A memory leak is a term used to describe error conditions that arise when programs allocate memory that is not reclaimed. As described above, garbage collection reclaims memory after the last reference to the memory is no longer "reachable" from a known set of "roots," wherein the roots include things such as processor registers and memory locations containing references to heap objects, such as those used

to hold the program stacks. Consequently, garbage collection may leave objects in the heap that are irrelevant (i.e., no longer needed) to the future of the computation but are still reachable from at least one object.

Some of these reachable but irrelevant objects arise from local variables that exist in the activation records of a program. The activation record includes variables (i.e., objects references) that are passed to a procedure from another procedure or routine as well as local variables that are declared within the program. These procedures may be long-lived and include memory hungry processing. Therefore, freeing objects that are no longer used is often vital to avoiding memory problems.

One approach to freeing objects is for the programmer to explicitly nullify a variable that will no longer be used in the program. A first challenge to this approach is that it nullifies the benefits of using garbage collection, as the programmer is forced to consider memory management issues even when such considerations may not be necessary. This is not a common practice with programmers of middleware applications, except when memory problems arise. A second challenge is completeness. Namely, identifying all nullifiable variables may be difficult and time consuming for a programmer. A third challenge is correctness: when attempting to nullify the variables, the programmer may introduce errors by prematurely nullifying a variable.

A fourth challenge regards single-assignment semantics. More specifically, a programmer may use

variables and parameters with single-assignment semantics, such as using the "const" keyword in the C programming language, or using the "final" attribute in a Java program. Single-assignment semantics are used to prevent certain types of programming errors as well as to communicate a variable's use clearly to other programmers and the compiler. A programmer desiring to use single-assignment cannot use nullification, as the nullification statement will be seen by the compiler as a second assignment, violating the single-assignment semantic. The programmer must therefore choose, for every variable, whether to use single-assignment or variable nullification, thus losing the benefits of using both techniques simultaneously.

Another challenge is that a compiler may actually be responsible for setting up a variable to reference an object as well as including the variable in the activation record of a program. In such situations, the programmer would not know the name of the variable or object being passed as a result of the compiler and, therefore, would not be able to include a statement nullifying the object or variable after it is no longer used.

Figure 1 is a prior art diagram showing how prior art compilers, including Just-in-Time (JIT) compilers, do not nullify variables after the last use of such variables. Garbage collector **100** is a process (i.e., a software routine) that reclaims memory from heap **110**. When invoked, garbage collector **100** determines which objects residing in the heap are no longer accessible ("reachable") from other objects ("roots"), such as the program stack **125**. Program stack **125** includes a start of the procedure **130**, often

identified by a left brace ({). The program stack includes activation records **140** that include parameters that are passed to the program as well as locally declared variables **150**. Variables included in the activation records refer to objects that are stored in memory that is included in heap **110**. Program stack **125** also includes statements that use objects and variables **160**. Such statements may retrieve data from objects and variables, store data in the objects and variables, or use the objects and variables in computations or to call other programs or procedures.

At various points in the program, a last use of a variable that references an object is made. One of these points is statement **170**. While only one point is shown in program stack **125**, it will be understood by those skilled in the art that many such points may exist in a given program when particular variables are no longer used (i.e., some variables may no longer be used while other variables continue to be used). Even though the variables are no longer used, program stack **125** continues to reference the unused variables. The continued reference by such variables to such no-longer-used objects results in garbage collector **100** maintaining (i.e., not removing) the memory used by the no-longer-used objects within heap **110**, notwithstanding the fact that there may be no objects needing the objects and variables. In the example shown, program stack **125** performs memory hungry computation **180**. Even if the middleware environment, such as a Java Virtual Machine, needs additional memory to perform the computation, it is unable to identify the no-longer-used objects as "garbage" until after the computation is

performed (i.e., at the end of the procedure **180** which is often identified with a closing brace (})).

What is needed, therefore, is a system and method for automatically nullifying variables that are no longer used by a program. What is further needed is a modification of the compiler to automatically nullify variables both referenced by the programmer as well as those included by the compiler.

SUMMARY

It has been discovered that the aforementioned challenges are resolved using a system and method that nullifies variables after they are no longer used. Once the variables are nullified, the space occupied by the objects that they formerly referenced in the heap can be reclaimed, if needed, by a garbage collection routine. Without nullifying the variables, the garbage collection routine must assume that the program is still using the objects and may not reclaim the space.

Before program execution, objects and variables to be included in activation records are compared with program statements to determine the last usage of such objects and variables. When an object or variable is last used in the program, a statement is automatically included in the program code nullifying the variable. In one embodiment, nullifying the variable is performed by setting the address of the referenced object to "null" (e.g., "object1 = null;"). Setting the object or variable to null destroys the reference of the program to the object residing in the heap. If that variable held the last reference to the object, then, after the program's reference to the object or variable has been nullified, the garbage collector is free to reclaim the memory used to store the object in the heap.

The foregoing is a summary and thus contains, by necessity, simplifications, generalizations, and omissions of detail; consequently, those skilled in the art will appreciate that the summary is illustrative only and is not

intended to be in any way limiting. Other aspects, inventive features, and advantages of the present invention, as defined solely by the claims, will become apparent in the non-limiting detailed description set forth below.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

Figure 1 is a prior art diagram showing how prior art compilers do not nullify variables and objects after the last use of such variables and objects;

Figure 2 is a diagram showing the nullification of variables and objects using a compiler;

Figure 3 is a flowchart showing additional steps taken by a compiler to add nullification statements to a JIT-compiled program;

Figure 4 is a JAVA example showing a nullification statement being added to a program by a Java Just-in-Time (JIT) compiler; and

Figure 5 is a block diagram of a computing device capable of implementing the present invention.

DETAILED DESCRIPTION

The following is intended to provide a detailed description of an example of the invention and should not be taken to be limiting of the invention itself. Rather, any number of variations may fall within the scope of the invention, which is defined in the claims following the description.

Figure 1 is a prior art diagram showing how prior art compilers do not nullify variables and objects after the last use of such variables and objects. For details concerning this diagram, see the "background" section, above.

Figure 2 is a diagram showing the nullification of variables that reference objects stored in a garbage collected heap. The components used and processing shown in **Figure 2** are similar to those shown in **Figure 1**, however in **Figure 2** variables are nullified by the compiler so that the memory they use in the heap might be reclaimed by the garbage collector.

Garbage collector **200** is a process (i.e., a software routine) that reclaims memory occupied by objects that are stored in heap **210**. When invoked, garbage collector **200** determines which objects residing in the heap are no longer accessible ("reachable") from other objects ("roots"), such as program stack **225**. Program stack **225** includes a start of the procedure **230**, often identified by a left brace ({). The program stack includes activation records **240** that include parameters that are passed to the program as well as locally declared variables **250**. Some of the variables

included in the activation records reference objects that are stored in garbage collected heap **210**. Program stack **225** also includes statements that use objects and variables **260**. Such statements may retrieve data from objects, store data in the objects, or call the objects to perform computations or other functions.

At various points in the program, a last use of a variable that references an object is made. One of these points is statement **270**. While only one point is shown in program stack **225**, it will be understood by those skilled in the art that many such points may exist in a given program when particular variables are no longer used (i.e., some variables may no longer be used while other variables continue to be used). After a variable is last-used by the program, the compiler inserts nullification statement **275** into the compiled program. In this manner, after the nullification statement, program stack **225** no longer references the objects that are no longer used by the program. The removal of the reference to the no-longer-used objects results in garbage collector **200** being able to remove the memory used by the no-longer-used objects from heap **210**, if there are no other programs needing the objects and variables. In the example shown, program stack **225** performs memory hungry computation **280**. If the middleware environment, such as a Java Virtual Machine, needs additional memory to perform the computation, it is possible that garbage collector **200** will be able to identify the no-longer-used objects as "garbage" before the computation is performed, thus freeing memory in the heap that might be needed by the memory hungry computation. Nullification statements are inserted by the compiler into

program stack **225** at appropriate points between the activation records and the end of the program **290**, which is often indicated by a closing brace (}).

Figure 3 is a flowchart showing additional steps taken by a compiler, such as a Java "Just-in-Time" ("JIT") compiler, to add nullification statements to a compiled program. Those skilled in the art will realize that the compiler performs more functions than shown in **Figure 3** and that **Figure 3** focuses on the processing added to the compiler to add nullification statements to a program.

Compiler processing commences at **300** whereupon, at step **305**, the compiler reads activation records **315** from source program **310**. Those skilled in the art will appreciate that source program **310** may be a program written by a programmer as well as a program that has had activation records inserted in the program by the compiler. The compiler records the variables that are included in the activation records.

The compiler reads, at step **320**, program statement **325** included in source program **310**. Program statements included in source program **310** may use include variables that were identified in the activation records, and these variables may reference objects stored in the heap. The compiler performs compilation steps on the statement and, at step **330**, writes the resulting compiled statement to resulting code **340** as program statement **350**.

The compiler then identifies, at step **355**, variables used in the most recently read program statement that are not used in subsequent statements found in source program **310**. A determination is made as to whether the most

recently read program statement is the "last use" of any of the variables (decision **360**). If the statement is the "last use" of any such variables, decision **360** branches to "yes" branch **362** whereupon, at step **365**, nullification statement **370** is written for those variables that are no longer used. An example, in the Java programming language, of a nullification statement that nullifies variable "Var1" that previously referenced object "ObjA" is "Var1 = null;" - the effect of this statement is to set the address of the object referenced by Var1 to a "null" value (i.e., no address) which severs the program's connection to the actual object ("Obj1") stored in the heap. When garbage collection occurs, the program will no longer be seen as a "root" of the object. If there are no other "roots" of the object, then the garbage collector is free to remove the object from the heap and reclaim the memory that it occupied.

On the other hand, if the statement is not the "last use" of any of the variables, then decision **360** branches to "no" branch **368** bypassing step **365**.

A determination is made as to whether there are more statements to process in source program **310** (decision **380**). If there are additional statements to process, decision **380** branches to "yes" branch **382** which loops back to read and process the next statement resulting in program statement **375** being written to resulting code **340**. Throughout this looping, when a "last use" of one or more variables is encountered, a nullification statement is written to resulting code **340** for such statements. Eventually, program end statement **385** is read from source program **310**, end statement **390** is written to resulting code **340**, and

decision **380** branches to "no" branch **392** whereupon compiler processing ends at **395**.

Figure 4 is a Java example showing a nullification statement being added to a program by the Java Just-in-Time (JIT) compiler. Listing **400** includes the program statements that existed prior to processing the program using JIT compiler **450**, the processing of which is described in detail in **Figure 3**.

Listing **400** includes activation records **410** and **420**. Activation record **410** is the argument statement passing arguments param1, param2, and param3 to the program. Activation record **420** shows the declaration of local variables value1 and value2. The comment indicates that no further use is made of param1 or value1 in the remainder of the program prior to the execution of statement **430**. Statement **430** is shown to be a "memory hungry" computation which may need more memory than is currently free in the JVM heap, causing a garbage collection event to occur in order to free space for the computation.

Listing **460** shows the same listing as shown in listing **400** after JIT compiler **450** automatically inserted nullification statements. Listing **460** includes nullification statement **470** which nullifies param1 and value1 (i.e., sets their value (referenced address) to null indicating that the variable is no longer referencing an object in the heap). Now, when memory hungry computation statement **430** executes, if the garbage collector needs to locate additional memory for the computation then the memory previously occupied by the objects referenced by param1 and value1 may be able to be reclaimed (so long as

no other program/object is currently a root for either object) providing additional memory for the execution of memory hungry computation **430**.

Figure 5 illustrates information handling system **501** which is a simplified example of a computer system capable of performing the computing operations described herein. Computer system **501** includes processor **500** which is coupled to host bus **502**. A level two (L2) cache memory **504** is also coupled to host bus **502**. Host-to-PCI bridge **506** is coupled to main memory **508**, includes cache memory and main memory control functions, and provides bus control to handle transfers among PCI bus **510**, processor **500**, L2 cache **504**, main memory **508**, and host bus **502**. Main memory **508** is coupled to Host-to-PCI bridge **506** as well as host bus **502**. Devices used solely by host processor(s) **500**, such as LAN card **530**, are coupled to PCI bus **510**. Service Processor Interface and ISA Access Pass-through **512** provides an interface between PCI bus **510** and PCI bus **514**. In this manner, PCI bus **514** is insulated from PCI bus **510**. Devices, such as flash memory **518**, are coupled to PCI bus **514**. In one implementation, flash memory **518** includes BIOS code that incorporates the necessary processor executable code for a variety of low-level system functions and system boot functions.

PCI bus **514** provides an interface for a variety of devices that are shared by host processor(s) **500** and Service Processor **516** including, for example, flash memory **518**. PCI-to-ISA bridge **535** provides bus control to handle transfers between PCI bus **514** and ISA bus **540**, universal serial bus (USB) functionality **545**, power management functionality **555**, and can include other functional

elements not shown, such as a real-time clock (RTC), DMA control, interrupt support, and system management bus support. Nonvolatile RAM **520** is attached to ISA Bus **540**. Service Processor **516** includes JTAG and I2C busses **522** for communication with processor(s) **500** during initialization steps. JTAG/I2C busses **522** are also coupled to L2 cache **504**, Host-to-PCI bridge **506**, and main memory **508** providing a communications path between the processor, the Service Processor, the L2 cache, the Host-to-PCI bridge, and the main memory. Service Processor **516** also has access to system power resources for powering down information handling device **501**.

Peripheral devices and input/output (I/O) devices can be attached to various interfaces (e.g., parallel interface **562**, serial interface **564**, keyboard interface **568**, and mouse interface **570** coupled to ISA bus **540**. Alternatively, many I/O devices can be accommodated by a super I/O controller (not shown) attached to ISA bus **540**.

In order to attach computer system **501** to another computer system to copy files over a network, LAN card **530** is coupled to PCI bus **510**. Similarly, to connect computer system **501** to an ISP to connect to the Internet using a telephone line connection, modem **575** is connected to serial port **564** and PCI-to-ISA Bridge **535**.

While the computer system described in **Figure 5** is capable of executing the processes described herein, this computer system is simply one example of a computer system. Those skilled in the art will appreciate that many other computer system designs are capable of performing the processes described herein.

One of the preferred implementations of the invention is a client application, namely, a set of instructions (program code) in a code module that may, for example, be resident in the random access memory of the computer. Until required by the computer, the set of instructions may be stored in another computer memory, for example, in a hard disk drive, or in a removable memory such as an optical disk (for eventual use in a CD ROM) or floppy disk (for eventual use in a floppy disk drive), or downloaded via the Internet or other computer network. Thus, the present invention may be implemented as a computer program product for use in a computer. In addition, although the various methods described are conveniently implemented in a general purpose computer selectively activated or reconfigured by software, one of ordinary skill in the art would also recognize that such methods may be carried out in hardware, in firmware, or in more specialized apparatus constructed to perform the required method steps.

While particular embodiments of the present invention have been shown and described, it will be obvious to those skilled in the art that, based upon the teachings herein, that changes and modifications may be made without departing from this invention and its broader aspects. Therefore, the appended claims are to encompass within their scope all such changes and modifications as are within the true spirit and scope of this invention. Furthermore, it is to be understood that the invention is solely defined by the appended claims. It will be understood by those with skill in the art that if a specific number of an introduced claim element is intended, such intent will be explicitly recited in the claim, and in

the absence of such recitation no such limitation is present. For non-limiting example, as an aid to understanding, the following appended claims contain usage of the introductory phrases "at least one" and "one or more" to introduce claim elements. However, the use of such phrases should not be construed to imply that the introduction of a claim element by the indefinite articles "a" or "an" limits any particular claim containing such introduced claim element to inventions containing only one such element, even when the same claim includes the introductory phrases "one or more" or "at least one" and indefinite articles such as "a" or "an"; the same holds true for the use in the claims of definite articles.